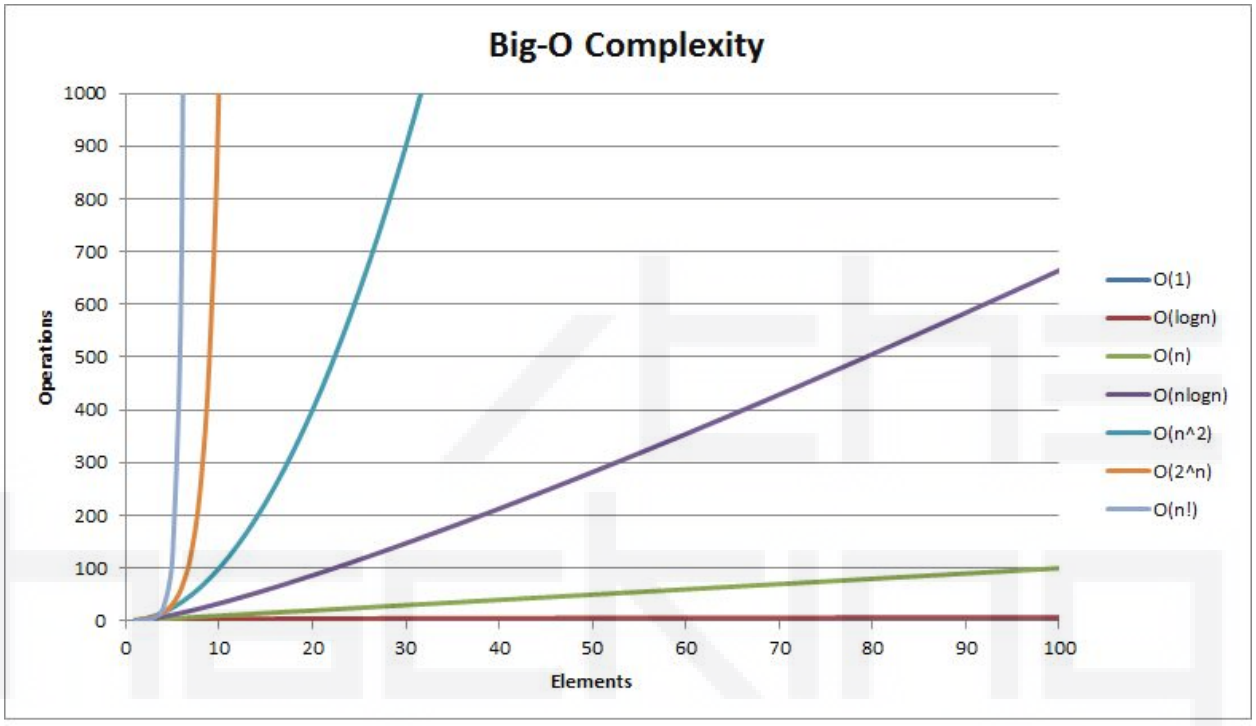
Algorithm Analysis - Time Complexity

Before understanding time complexity, arrange the following math functions in ascending order.

Math Functions :

f(1), f(log(n)), f(n), f(n log n), $f(n^2)$, $f(n^3)$, $f(2^n)$, f(n!)

Discuss the above math functions and arrange them in ascending/descending order. Discuss with your instructor about the order of growth. We are going to measure the algorithm performance and growth using the above mathematical functions.



Why do we need to learn performance analysis?

There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?

The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!

To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1) It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs the second performs better.

2) It might also be possible that for some inputs, the first algorithm performs better on one machine and the second works better on another machine for some other inputs.

Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate how the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search algorithm problem that we implemented last week(searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and the other way is Binary Search (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer A and Binary Search on a slow computer B and we pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds.

Let's say the constant for A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B. For small values of input array size n, the fast computer may take less time. But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine.

The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after a certain value of input size.

Here are some running times for this example:(approximate)

Linear Search running time in seconds on A: 0.2 * n
Binary Search running time in seconds on B: 1000*log(n)

Asymptotic analysis overcomes the problems of naive way of analyzing algorithms. In this post, we will take an example of Linear Search and analyze it using Asymptotic analysis.

We can have three cases to analyze an algorithm:
1) Worst Case
2) Average Case
3) Best Case

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be **Θ(n).**

Average Case Analysis (Sometimes done)
In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). The value of average case time complexity
Θ(n)

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be
Θ(1)
Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Practice Exercises

Find the Time Complexity of the following pseudo code snippets.

Example 1

```
x = n
while ( x > 0 ) {
    x = x - 1
}
```

The above is O(n)

Example 2

```
x = n
 while ( x > 0 ) {
     x = x / 2
 }
```

The above is O(logn)

Example 3

```
x = n
 while ( x > 0 ) {
    y = n
```

```
    while ( y > 0 ) {
        y = y - 1
    }
    x = x - 1
}
```

The above is $O(n^2)$

Example 4

```
x = n
while ( x > 0 ) {
    y = x
    while ( y > 0 ) {
        y = y - 1
    }
    x = x - 1
}
```

The above is $O(n^2)$

Example 5

```
x = n
while ( x > 0 ) {
    y = n
    while ( y > 0 ) {
        y = y / 2
    }
    x = x - 1
}
```

The above is O(nlogn)

Example 6

```
x = n
while ( x > 0 ) {
    y = n
    while ( y > 0 ) {
        y = y - 1
    }
    x = x / 2
}
```

The above is O(nlogn)

Example 7

```
var a = 0, b = 0;
```

```
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

The above is O(N + M)

Example 8

```
var a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

The above is $O(N^2)$

Example 9

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

The above is O(n log n)

Example 10

```
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

The above is O(logn)

Note : Not down the time complexities of the algorithms that you solved so far and discuss the complexity analysis of them with your instructor.

Asynchronous Programming with Javascript(Understanding Event Loop)

Blocking (Synchronous)
Blocking/Synchronous is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a blocking operation is occurring.

In Node.js/v8 engine, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as blocking. Synchronous methods in the Node.js standard library that use libuv are the most commonly used blocking operations. Native modules may also have blocking methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are non-blocking, and accept callback functions. Some methods also have blocking counterparts, which have names that end with Sync.

Synchronous vs Asynchronous

Synchronous (or sync) execution usually refers to code executing in sequence. In sync programming, the program is executed line by line, one line at a time. Each time a function is called, the program execution waits until that function returns before continuing to the next line of code.

Asynchronous (or async) execution refers to execution that doesn't run in the sequence it appears in the code. In async programming the program doesn't wait for the task to complete and can move on to the next task.

In the following example, the sync operation causes the alerts to fire in sequence. In the async operation, while alert(2) appears to execute second, it doesn't.

```
// Synchronous: 1,2,3
alert(1);
alert(2);
alert(3);

// Asynchronous: 1,3,2
alert(1);
setTimeout(() => alert(2), 0);
alert(3);
```

An async operation is often I/O related, although setTimeout is an example of something that isn't I/O but still async, which will be carried out by Event Loop in Web Browser or Node JS runtime environment in Node JS with the help of C/C++ underhood mechanism . Generally speaking, anything computation-related is sync and anything input/output/timing-related is async. The reason for I/O operations to be done asynchronously is that they are very slow and would block further execution of code otherwise.

Check out this Animation to understand the callbacks of async operations.
http://latentflip.com/loupe/

Lets try a few more exercises to understand asynchronous operations with the help of setTimeout method. Carefully identify the order of execution of each and every code snippet below. Discuss with your instructor to understand the below exercise event loop executions.

Exercise 1

```
function getUser() {
   setTimeout(() => {
       const userids = [10, 20, 30, 40];
       console.log(userids);
       setTimeout((id) => {
           const user = {
               name: 'John Doe',
               age: 25
           };
           console.log(`User ID : ${id} : User name : ${user.name}, User
Age: ${user.age}`);
           setTimeout((age) => {
               console.log(user);
           }, 1000, user.age);
       }, 1000, userids[3]);
   }, 1500)
}
getUser();
```

Output
[10, 20, 30, 40]
User ID : 40 : User name : John Doe, User Age: 25
{ name: 'John Doe', age: 25 }

Exercise 2

```
const arr = [10, 12, 15, 21];
for (var i = 0; i < arr.length; i++) {
  setTimeout(function() {
    console.log('Index: ' + i + ', element: ' + arr[i]);
  }, 3000);
}
```

Output
Index: 4, element: undefined
Index: 4, element: undefined
Index: 4, element: undefined
Index: 4, element: undefined

Exercise 3

```
const arr = [10, 12, 15, 21];
```

```
for (let i = 0; i < arr.length; i++) {
  setTimeout(function() {
    console.log('Index: ' + i + ', element: ' + arr[i]);
  }, 3000);
}
```

Output
Index: 0, element: 10
Index: 1, element: 12
Index: 2, element: 15
Index: 3, element: 21

Exercise 4

```
function getUser() {
    setTimeout(function(){
        const userids = [10, 20, 30, 40];
        console.log(userids);
        setTimeout(function(id){
            const user = {
                name: 'John Doe',
                age: 25
            };
            console.log("User ID : "+id +": User name : "+user.name+", User
Age: "+user.age);
            setTimeout(function(age){
                console.log(user);
            }, 1000, user.age);
        }, 1000, userids[3]);
    }, 1500)
}

 function abc(str,cb){
     console.log(str);
     cb();
 }
 function def(){
     console.log("I am deaf");
 }
 abc("prash",getUser);
 def();
```

Output
prash
I am deaf
[10, 20, 30, 40]
User ID : 40: User name : John Doe, User Age: 25
{ name: 'John Doe', age: 25 }

Exercise 5

```
function getUser() {
    setTimeout(function () {
        const userids = [10, 20, 30, 40];
        console.log(userids);
        setTimeout(function (id) {
            const user = {
                name: 'John Doe',
                age: 25
            };
            console.log("User ID : " + id + ": User name : " + user.name +
", User Age: " + user.age);
            setTimeout(function (age) {
                console.log(user);
            }, 1000, user.age);
        }, 1000, userids[3]);
    }, 1500)
}
function abc(str, cb) {
    setTimeout(() => {
        cb();
        console.log(str);
    }, 0)
}
function def() {
    console.log("I am from def");
}
function efg() {
    console.log('I am from efg');
}
abc("prash", getUser);
def();
def();
def();
def();
def();
def();
setTimeout(() => { efg(), 1000 });
def();
def();
def();
def();
def();
def();
setTimeout(() => { def(), 0 });
def();
def();
def();
def();
```

Output

I am from def

I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
I am from def
prash
I am from efg
I am from def
[10, 20, 30, 40]
User ID : 40: User name : John Doe, User Age: 25
{ name: 'John Doe', age: 25 }

Introduction to Node JS

Node.js is a platform that executes server-side JavaScript programs that can communicate with I/O sources like networks and file systems.

When Ryan Dahl created Node in 2009 he argued that I/O was being handled incorrectly, blocking the entire process due to synchronous programming.

Traditional web-serving techniques use the thread model, meaning one thread for each request. Since in an I/O operation the request spends most of the time waiting for it to complete, intensive I/O scenarios entail a large amount of unused resources (such as memory) linked to these threads. Therefore the “one thread per request” model for a server doesn’t scale well.

Dahl argued that software should be able to multi-task and proposed eliminating the time spent waiting for I/O results to come back. Instead of the thread model, he said the right way to handle several concurrent connections was to have **a single-thread, an event loop and non-blocking I/Os**. For example, when you make a query to a database, instead of waiting for the response you give it a callback so your execution can run through that statement and continue doing other things. When the results come back you can execute the callback.

The event loop is what allows Node.js to perform non-blocking I/O operations despite the fact that **JavaScript is single-threaded**. The loop, which runs on the same thread as the JavaScript code, grabs a task from the code and executes it. If the task is async or an I/O operation the loop offloads it to the system kernel, like in the case for new connections to the server, or to a thread pool, like file system related operations. The loop then grabs the next task and executes it.

Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes (this is an event), the kernel tells Node.js so that the appropriate callback (the one that depended on the operation completing) may be added to the poll queue to eventually be executed.

Node keeps track of unfinished async operations, and the event loop keeps looping to check if they are finished until all of them are.

To accommodate the single-threaded event loop, Node.js uses the **libuv** http://libuv.org/ library, which, in turn, uses a fixed-sized thread pool that handles the execution of some of the non-blocking asynchronous I/O operations in parallel. The main thread call functions post tasks to the shared task queue, which threads in the thread pool pull and execute.

Inherently non-blocking system functions such as networking translate to kernel-side non-blocking sockets, while inherently blocking system functions such as file I/O run in a blocking way on their own threads. When a thread in the thread pool completes a task, it informs the main thread of this, which in turn, wakes up and executes the registered callback.

https://nodejs.org/en/about/

Node Package Manager (NPM)

Node Package Manager (NPM) provides two main functionalities −

- Online repositories for node.js packages/modules which are searchable on https://www.npmjs.com/
- Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

NPM comes bundled with Node.js installation. To verify the same, open console and type the following command and see the result −

```
$ npm --version
```

Lets create a Javascript project

```
prash@PRASH-NITRO5:~$ mkdir js_proj
prash@PRASH-NITRO5:~$ cd js_proj/
prash@PRASH-NITRO5:~/js_proj$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.
```

```
Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (js_proj) cmd-app
version: (1.0.0)
description: Building Command line Applications with Javascript
entry point: (index.js) index.js
test command:
git repository:
keywords: node
author: School Of Coding
license: (ISC)
About to write to /home/prash/js_proj/package.json:

{
  "name": "cmd-app",
  "version": "1.0.0",
  "description": "Building Command line Applications with
Javascript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node"
  ],
  "author": "School Of Coding",
  "license": "ISC"
}


Is this OK? (yes) yes
prash@PRASH-NITRO5:~/js_proj$
```

The above **npm init** command will setup a javascript project in the folder **js_proj** leaves a file called **package.json**

```
prash@PRASH-NITRO5:~/js_proj$ ls
package.json
prash@PRASH-NITRO5:~/js_proj$ cat package.json
{
```

```
  "name": "cmd-app",
  "version": "1.0.0",
  "description": "Building Command line Applications with
Javascript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node"
  ],
  "author": "School Of Coding",
  "license": "ISC"
}
prash@PRASH-NITRO5:~/js_proj$
```

Attributes of Package.json

- name − name of the project
- version − version of the project
- description − description of the project
- author − author of the project
- dependencies − list of dependencies/modules required to run the project
- repository − repository type and URL of the package of where project is hosted
- main − entry file point of the project
- keywords − keywords to describe the tech specs of the project

Command Line Interface

A command-line interface (CLI) processes commands to a computer program in the form of lines of text. The program which handles the interface is called a command-line interpreter or command-line processor.

A command line interface (CLI) is a text-based user interface (UI) used to view and manage computer files. Command line interfaces are also called command-line user interfaces, console user interfaces and character user interfaces.

Before the mouse, users interacted with an operating system (OS) or application with a keyboard. Users typed commands in the command line interface to run tasks on a computer.

Today, most users prefer the graphical user interface (GUI) offered by operating systems such as Windows, Linux and MacOS. Most current Unix-based systems offer both a command line interface and a graphical user interface.

Command Line Inputs

Learning how to work with command line inputs is part of CLI programming. Command Line Inputs are strings/numbers that you enter while interacting with your program. Typically command line inputs are used to enter input data to perform certain operations required by that application.

In Javascript, we achieve this with Node JS (Javascript v8 Runtime environment) to process command line inputs. You cannot achieve this with a web browser.

In Node JS we take command line inputs with the help of a process , built in a module that requires understanding of asynchronous programming. So we install an external module named **readline-sync** to process command line inputs.

Exercise

Develop a program to sum two numbers whereas the input numbers are taken by command line.

Solution

Step 1 : Create a JS Project

```
prash@PRASH-NITRO5:~$ mkdir js_proj
prash@PRASH-NITRO5:~$ cd js_proj/
prash@PRASH-NITRO5:~/js_proj$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.
```

```
See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (js_proj) cmd-app
version: (1.0.0)
description: Building Command line Applications with Javascript
entry point: (index.js) index.js
test command:
git repository:
keywords: node
author: School Of Coding
license: (ISC)
About to write to /home/prash/js_proj/package.json:

{
  "name": "cmd-app",
  "version": "1.0.0",
  "description": "Building Command line Applications with
Javascript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node"
  ],
  "author": "School Of Coding",
  "license": "ISC"
}


Is this OK? (yes) yes
prash@PRASH-NITRO5:~/js_proj$
```

Step 2 : Install **readline-sync** module

https://www.npmjs.com/package/readline-sync

readline-sync module accepts command line inputs synchronously for interactively running to have a conversation with the user via a console(TTY).

```
$ npm i readline-sync --save
```

After this operation you will find an additional field in your package.json file. Additionally it will also create a new folder called **node_modules** in the project folder where the **readline-sync** module related source code exists.

```
$ cat package.json
{
  "name": "cmd-app",
  "version": "1.0.0",
  "description": "Building Command line Applications with Javascript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node"
  ],
  "author": "School Of Coding",
  "license": "ISC",

 "dependencies": {
    "readline-sync": "^1.4.10"
  }
}
prash@PRASH-NITRO5:~/js_proj$
```

Step 3 : Create an entry file for the project **index.js**

```
$ touch index.js
```

index.js

```
const readlineSync = require('readline-sync');
const num1 = Number(readlineSync.question("Enter the First Number :"));
const num2 = Number(readlineSync.question("Enter the Second Number :"));

function sum(a, b) {
    console.log(a + b);
}
sum(num1, num2);
```

package.json

```
"scripts": {
```

```
    "start": "node index.js"
 },
```

Step 4 : Run the script

```
$ npm start
```

Output :

```
prash@PRASH-NITRO5:~/js_proj$ npm start

> cmd-app@1.0.0 start /home/prash/js_proj
> node index.js

Enter the First Number :11
Enter the Second Number :3
14
prash@PRASH-NITRO5:~/js_proj$
```

Resources

https://www.npmjs.com/package/readline-sync

Command Line Arguments

Passing in arguments via the command line is an extremely basic programming task, and a necessity for anyone trying to write a simple Command-Line Interface (CLI).

Note: Unlike Command Line Inputs, Command Line Arguments are given before executing the program

Example :

```
prash@PRASH-NITRO5:~/js_proj$ node index.js 11 3
```

Here **11** and **3** are command line arguments to index.js file

All command-line arguments received by the shell are given to the process in an array called **argv** (short for 'argument values').

```
console.log(process.argv);
```

Exercise :

Demonstrate how to take command line arguments with Javascript.

Step 1 : Create a JS file **arguments.js**

```
$ node arguments.js one two three four five
```

arguments.js

```
console.log(process.argv);
```

Output

```
$ node arguments.js one two three four five
[ 'node',
  '/home/avian/argvdemo/argv.js',
  'one',
  'two',
  'three',
  'four',
  'five' ]
```

There you have it - an array containing any arguments you passed in. **Notice the first two elements - node and the path to your script**. These will always be present - even if your program takes no arguments of its own, your script's interpreter and path are still considered arguments to the shell you're using.

Where everyday CLI arguments are concerned, you'll want to skip the first two. Now try this in **arguments.js**:

```
var myArgs = process.argv.slice(2);
console.log('myArgs: ', myArgs);
```

Output

```
$ node arguments.js one two three four
myArgs:  [ 'one', 'two', 'three', 'four' ]
```

Example Program 1

```
var myArgs = process.argv.slice(2);
console.log('myArgs: ', myArgs);

switch (myArgs[0]) {
case 'insult':
    console.log(myArgs[1], 'smells quite badly.');
    break;
case 'compliment':
    console.log(myArgs[1], 'is really cool.');
    break;
default:
    console.log('Sorry, that is not something I know how to do.');
}
```

Exercise

Develop a program to sum two numbers whereas the input numbers are taken by command line arguments.

code.js

```
var myArgs = process.argv.slice(2);
console.log('myArgs: ', myArgs);

function sum(a, b) {
    console.log(Number(a) + Number(b));
}
sum(myArgs[0], myArgs[1]);
```

Output

```
$ node code.js 5 5

myArgs:  [ '5', '5' ]
10
prash@PRASH-NITRO5:~/js_proj$
```

Building Menu Driven CLI App with Javascript

Exercise 1

Building Menu Driven CLI app with Javascript that will sum two number taken via command line inputs

Step 1 : Create a JS Project

```
prash@PRASH-NITRO5:~$ mkdir js_proj
prash@PRASH-NITRO5:~$ cd js_proj/
prash@PRASH-NITRO5:~/js_proj$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (js_proj) cmd-app
version: (1.0.0)
description: Building Command line Applications with Javascript
entry point: (index.js) index.js
test command:
git repository:
keywords: node
author: School Of Coding
license: (ISC)
About to write to /home/prash/js_proj/package.json:

{
  "name": "cmd-app",
  "version": "1.0.0",
  "description": "Building Command line Applications with
Javascript",
```

```
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node"
  ],
  "author": "School Of Coding",
  "license": "ISC"
}


Is this OK? (yes) yes
prash@PRASH-NITRO5:~/js_proj$
```

Step 2 : Install **readline-sync** module

https://www.npmjs.com/package/readline-sync

readline-sync module accepts command line inputs synchronously for interactively running to have a conversation with the user via a console(TTY).

```
$ npm i readline-sync --save
$ npm i axios --save
```

After this operation you will find an additional field in your package.json file. Additionally it will also create a new folder called **node_modules** in the project folder where the **readline-sync** module related source code exists.

```
$ cat package.json
{
  "name": "cmd-app",
  "version": "1.0.0",
  "description": "Building Command line Applications with Javascript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node"
  ],
  "author": "School Of Coding",
  "license": "ISC",

 "dependencies": {
    "readline-sync": "^1.4.10",
  }
```

```
}
prash@PRASH-NITRO5:~/js_proj$
```

Step 3 : Create an entry file for the project **index.js**

```
$ touch index.js
```

index.js

```
const readlineSync = require('readline-sync');

function sum(a, b) {
    return a + b;
}

function cli() {
    console.clear();
    const num1 = Number(readlineSync.question('Enter First  Number : '));
    const num2 = Number(readlineSync.question('Enter Second Number : '));
    console.log(sum(num1, num2));

 const repeat = readlineSync.question('Do you want to add another number (y/n)? ');
    if (repeat === 'y') {
        cli();
    } else {
        console.log("Bye Bye");
    }
}

cli();
```

package.json

```
"scripts": {
    "start": "node index.js"
 },
```

Step 4 : Run the script

```
$ npm start
```

Output:

```
prash@PRASH-NITRO5:~/js_proj$
Enter First  Number : 9
Enter Second Number : 9
```

```
18
Do you want to add another number (y/n)? n
Bye Bye
```

Exercise 2

Building Menu Driven CLI app with Javascript that fetches Weather Report in Console

Step 1 : Create a JS Project

```
prash@PRASH-NITRO5:~$ mkdir js_proj
prash@PRASH-NITRO5:~$ cd js_proj/
prash@PRASH-NITRO5:~/js_proj$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (js_proj) cmd-app
version: (1.0.0)
description: Building Command line Applications with Javascript
entry point: (index.js) index.js
test command:
git repository:
keywords: node
author: School Of Coding
license: (ISC)
About to write to /home/prash/js_proj/package.json:

{
  "name": "cmd-app",
  "version": "1.0.0",
  "description": "Building Command line Applications with
Javascript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

```
  "keywords": [
    "node"
  ],
  "author": "School Of Coding",
  "license": "ISC"
}


Is this OK? (yes) yes
prash@PRASH-NITRO5:~/js_proj$
```

Step 2 : Install **readline-sync** and **axios** module

https://www.npmjs.com/package/readline-sync

readline-sync module accepts command line inputs synchronously for interactively running to have a conversation with the user via a console(TTY).

```
$ npm i readline-sync --save
$ npm i axios --save
```

After this operation you will find an additional field in your package.json file. Additionally it will also create a new folder called **node_modules** in the project folder where the **readline-sync** and **axios** module related source code exists.

```
$ cat package.json
{
  "name": "cmd-app",
  "version": "1.0.0",
  "description": "Building Command line Applications with Javascript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node"
  ],
  "author": "School Of Coding",
  "license": "ISC",

 "dependencies": {
    "readline-sync": "^1.4.10",
    "axios": "^0.19.2",
  }
}
prash@PRASH-NITRO5:~/js_proj$
```

Step 3 : Create an entry file for the project **index.js**

```
$ touch index.js
```

index.js

```
const readlineSync = require('readline-sync');
const axios = require('axios');
const getCityWeather = (cityName) => {
    const url = 'http://api.openweathermap.org/data/2.5/weather?q=';
    const apiCall = `${url}${cityName}&APPID=1cbef1c38b3372ffd59a3081142939b7`;
    return axios.get(apiCall);
}

function weatherApp() {
    console.clear();
    const cityName = readlineSync.question('Enter the City Name :  ');
    getCityWeather(cityName)
        .then(({ data: { main, name } }) => {
            const { temp, humidity } = main
            console.log(`=================================`);
            console.log(`|       ${name} Weather          |`);
            console.log(`=================================`);
            console.log(`|  Temperature   |  ${temp} K       |`);
            console.log(`|  Humidity      |  ${humidity}%          |`);
            console.log(`=================================`);

            const repeat = readlineSync.question('Do you want to get another city
weather (y/n)? ');
            if (repeat === 'y') {
                weatherApp();
            }else{
                console.log("Bye Bye");
            }
        })
        .catch((err) => {
            console.log(err);
            weatherApp();
        });
}

weatherApp();
```

package.json

```
"scripts": {
    "start": "node index.js"
 },
```

Step 4 : Run the script

```
$ npm start
```

Output:

```
Which city weather information you want to know? Goa
==================================
|       Goa Weather            |
==================================
|  Temperature    |  298.36 K       |
|  Humidity      |  90%           |
==================================
Do you want to get another city weather (y/n)? n
Bye Bye
```

SetTimeout Exercise with Node JS

```
function abc() {
      return 'ABC';
}
setTimeout(todo1, 3000);
console.log(123);
console.log(abc());
function todo1() {
      console.log('I am Doing1');
}
function todo2() {
      console.log('I am Doing2');
}
setTimeout(todo2, 3000);
console.log(def());
console.log('Hello There ');
console.log(123);
function def() {
      return 'DEF';
}
```

Output

123
ABC
DEF
Hello There
123

I am Doing1
I am Doing2

Node JS File System

Node implements File I/O using simple wrappers around standard POSIX functions. The Node File System (fs) module can be imported using the following syntax −

```
var fs = require("fs")
```

Example

```
const fs = require('fs');
console.log('I am about to open the file hai.txt');
//Read File Call Back
fs.readFile('hai.txt', (err, data) => {
      if (err) {
            throw err;
      }
      console.log(data.toString());
});
const data = fs.readFileSync('new.txt');
console.log(data.toString());
console.log('I am done reading the file hai.txt');
```

Resources FS Methods

https://www.tutorialspoint.com/nodejs/nodejs_file_system.htm

FS Exercise

Use Standard Inputs and Command Line Arguments for the below problem.

1) Create a program to take input data and file name through command line input
2) Create a file with the content from cmd line input(stdin) and save the file in dest
3) Read the content that is saved and print the buffer

Solution

```
const fs = require('fs');
const readLineSync = require('readline-sync');
const fileData = readLineSync.question('Please Enter your Data : \n');
const fileName = readLineSync.question('Please Enter File Name : ');
```

```
fs.writeFile(`/path/${fileName}`, fileData, (err) => {
      if (err) {
            throw err;
      } else {
            fs.readFile(`/path/${fileName}`, (err, data) => {
                  if (err) {
                        throw err;
                  } else {
                        console.log(data.toString());
                  }
            });
            console.log('File is succesfully Created');
      }
});
```

Function Callback leads to a problem called ‘CallBack Hell”

http://callbackhell.com/

Then there is Promise to avoid callback hell.

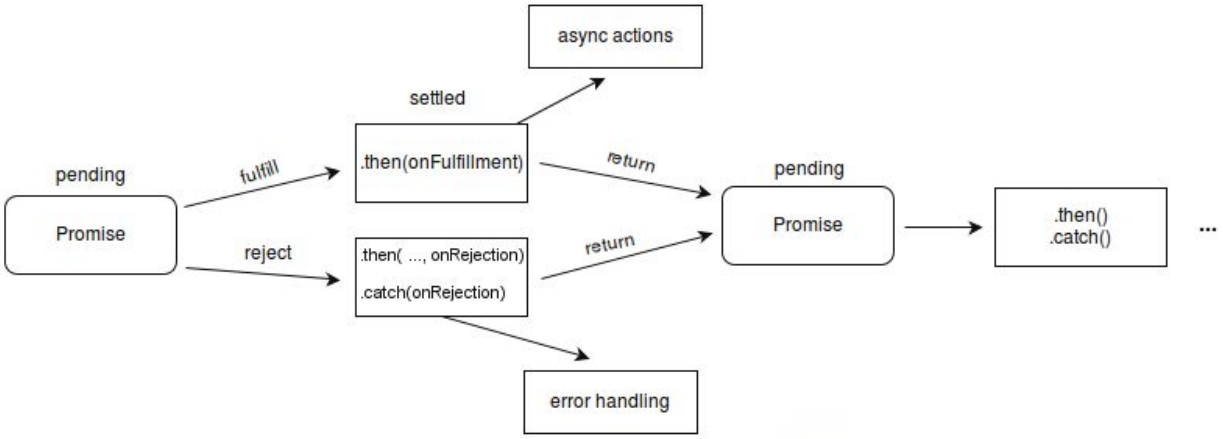
So far we have seen function callbacks to perform async operations. Lets start learning promises now

---

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

pending: initial state, neither fulfilled nor rejected.
fulfilled: meaning that the operation completed successfully.
rejected: meaning that the operation failed.
A pending promise can either be fulfilled with a value, or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.



Promise Exercise 1 :
Lets make axios call (GET) to 'https://api.ipify.org?format=json' => save the IP in text file.

Promise : Axios : To make Network requests

```
const axios = require('axios');
const fs = require('fs');
//HTTP GET

console.log('I m first');
axios
      .get('https://api.ipify.org?format=json')
      .then((response) => {
            console.log(response.data);
            fs.writeFile('ip.txt', response.data.ip, (err) => {
                  if (err) {
                        throw err;
                  }
                  console.log('Check the file');
            });
      })
      .catch((err) => {
            console.log(err);
      });

console.log('I m second');
// pending: initial state, neither fulfilled nor rejected.
// fulfilled: meaning that the operation completed successfully.
// rejected: meaning that the operation failed.
```

Promise Exercise 2 :
Read the above created ip.txt file using Node FS and demonstrate promise chaining example (Promise Chaining (V Imp))

```
const fs = require('fs');

function dodo() {
	return new Promise((resolve, reject) => {
		fs.readFile('ip.txt', (err, data) => {
			if (err) {
				reject(err);
			} else {
				resolve(data.toString());
			}
		});
	});
}
dodo()
	.then((data) => {
		console.log(data);
	})
	.catch((err) => {
		console.log(err);
	});
```

Promise Exercise 3 :
Note : Ask your instructor to set up a Node End API point that takes $URL/number1/number2 and returns the sum of the numbers.

Test API : If you ping URL/2/3 , the response should be 5.

Problem :

- Hit the above API with axios by giving the two numbers from command line as input
- After receiving the sum of the above two numbers from the API, take the third number from the command line input and hit the API again with sum and third number.
- Return the final sum of three numbers.
- Note : Take third number input only after receiving the sum of the first entered two numbers.
- This is another demonstration of promise chaining.

Solution (Method 1)

```
/*
GET http://localhost:5000/add/num1/num2
Reponse : 5
*/
const axios = require('axios');
```

```
const readLineSync = require('readline-sync');

const num1 = readLineSync.question('Enter Num1 : ');
const num2 = readLineSync.question('Enter Num2 : ');
axios
      .get(`http://localhost:5000/add/${num1}/${num2}`)
      .then((res) => {
            let sum1 = res.data.split(' ')[1];
            const num3 = readLineSync.question('Enter Num3 : ');
            axios
                  .get(`http://localhost:5000/add/${sum1}/${num3}`)
                  .then((res) => {
                        console.log(res.data);
                  })
                  .catch((err) => {
                        console.log(err);
                  });
      })
      .catch((err) => {
            console.log(err);
      });
```

Solution (Method 2) with extra number

```
const axios = require('axios');
const readLineSync = require('readline-sync');

const num1 = readLineSync.question('Enter Num1 : ');
const num2 = readLineSync.question('Enter Num2 : ');

function dosum(a, b) {
      return new Promise((resolve, reject) => {
            axios
                  .get(`http://localhost:5000/add/${num1}/${num2}`)
                  .then((res) => {
                        let sum = res.data.split(' ')[1];
                        resolve(sum);
                  })
                  .catch((err) => {
                        reject(err);
                  });
      });
}
dosum(num1, num2)
      .then((sum) => {
            const num3 = readLineSync.question('Enter Num3 : ');
            return Number(sum) + Number(num3);
      })
      .then((sum) => {
            const num4 = readLineSync.question('Enter Num4 : ');
```

```
            console.log(Number(sum) + Number(num4));
      });
```

Asynchronous Command Line Inputs without using readline-sync third party library

```
//Command Line Inputs / Standard Input Async
const rl = readline.createInterface({
      input: process.stdin,
      output: process.stdout
});
console.log("First");
rl.question('What is your name ? ', (name) => {
      rl.question('Enter your address : ', (address) => {
            console.log(name);
            console.log(address);
            rl.close();
      });
});
console.log("Second");
```

Argument object in JS

```
function fun(a, b, c) {
      if (arguments.length == 2) {
            console.log(a + b);
      }
      if (arguments.length == 3) {
            console.log(a * b * c);
      }
}

fun(2, 3);
fun(2, 3, 5);
```

Function Default Parameters Example

```
function fun(a=1,b=0){
    console.log(a,b);
}

fun(7,8)
```

```
//Function Default Parameters
```

Converting CallBacks to Promises (promisify utility)

Simple Callback Example that we are going to convert into Promise.

```
fs.readFile('ip.txt', (err, data) => {
      if (err) {
            throw err;
      }
      console.log(data.toString());
 });
```

Method 1 : Converting the above Function Callback to Promise

```
const fs = require('fs');
const util = require('util');
 function todo() {
      return new Promise((resolve, reject) => {
            fs.readFile('ip2.txt', (err, data) => {
                  if (err) {
                        reject(err);
                        return;
                  }
                  resolve(data.toString());
            });
      });
  }

  todo()
      .then((data) => {
            console.log(data);
      })
      .catch((err) => {
            console.log(err);
      });
```

Method 2 : Using Util.Promisify

```
const readFile = util.promisify(fs.readFile);

readFile('ip.txt')
      .then((data) => {
            console.log(data.toString());
      })
      .catch((err) => {
            console.log(err);
```

```
      });
```

Async Await

Convert the axios promise into async await

Usual Promise Example

```
const axios = require('axios');

   axios
      .get('https://jsonplaceholder.typicode.com/todos/1')
      .then((res) => {
            console.log(res.data);
      })
      .catch((err) => {
            console.log(err);
      });
```

The above promise can be converted into modern aysnc await syntax.

```
//Async Await
async function ping() {
      try {
            const res = await
axios.get('https://jsonplaceholder.typicode.com/todos/1');
            console.log(res.data);
      } catch (err) {
            console.log(err);
      }
}
ping();
```

Async await Example 2

```
const axios = require('axios');
//Async Await
console.log("Started");
(async () => {
      try {
            console.log(1);
            const res1 = await
axios.get('https://jsonplaceholder.typicode.com/todos/1');
```

```
            console.log(res1.data);
            console.log(3);

            const res2 = await
axios.get('https://jsonplaceholder.typicode.com/todos/2');
            console.log(res2.data);
      } catch (err) {
            console.log(err);
      }
})();
console.log("Done");
```

Converting CallBacks to Async await with (promisify utility)

Simple Callback Example that we are going to convert into async await code.

```
const fs = require('fs');
const util = require('util');
//Fcb to async await
const readFile = util.promisify(fs.readFile);

async function foo() {
      try {
            const data = await readFile('ip.txt');
            console.log(data.toString());
      } catch (err) {
            console.log(err);
      }
}
foo();
```

Generators
Regular functions return only one, single value (or nothing).

Generators can return (“yield”) multiple values, one after another, on-demand. They work great with iterables, allowing to create data streams with ease.

Generator functions
To create a generator, we need a special syntax construct: **function***, so-called “generator function”.

It looks like this:

```
function* generateSequence() {
```

```
  yield 1;
  yield 2;
  return 3;
}
```

Resource : https://javascript.info/generators

https://javascript.info/closure

Algorithms

Problem 1 : Given an array of positive integer elements.Print out the count of Prime Numbers and Narcissistic numbers from the array and remove those elements from the array.

Sample Input 1 :10,20,30,40,2,5,7,153,1024,1634,11
Expected Output format :
Prime Numbers Count : 4
Narcissistic Numbers Count : 5
The updated array : 10,20,30,40,1024
Note : The Inputs must be given via command line like we studied in the above exercises about readline-sync to accept command line inputs.

Solution

```
var readlineSync = require('readline-sync');
//O(sqrt(n)) Time Complexity
function isPrime(num) {
  for (let i = 2; i * i <= num; i++) {
    if (num % i == 0) {
      return false;
    }
  }
  return true && num !== 1;
}

//O(log n * log n)
function isNarcissistic(number) {
  let sum = 0;
  let result = false;
  let num = number;
  let power = Math.floor(Math.log10(number) + 1);

  while (num !== 0) {
    sum += Math.pow(num % 10, power);
```

```
    num = parseInt(num / 10);
  }
  if (number === sum) {
    result = true;
  }
  return result;
}

//O(sqrt(n)) + O((log n * log n)
function getPrimesAndNarcissistic(arr) {
  let primes = 0;
  let nars = 0;
  let result = [];
  let primeResult, narResult;

  arr.forEach((num) => {
    primeResult = isPrime(Number(num));
    narResult = isNarcissistic(Number(num));

    if (primeResult && narResult) {
      primes++;
      nars++;
    } else if (primeResult) {
      primes++;
    } else if (narResult) {
      nars++;
    } else {
      result.push(num);
    }
  });

  console.log('Prime Numbers Count : ', primes);
  console.log('Narcissistic Numbers Count : ', nars);
  console.log('The updated array : ', result);
}

function getInputs() {
  let takeInputs = true;
  while (takeInputs) {
    let arr = readlineSync.question('Enter comma separated array values :
');
    if (arr.length > 0) getPrimesAndNarcissistic(arr.split(','));
    let input = readlineSync.question(
      "Press 'N' to Terminate or 'Y' to continue : "
    );

    if (input === 'N' || input === 'n') {
      takeInputs = false;
    }
  }
}
```

```
getInputs();
```

Problem 2 : Create an interactive command line program that continually takes a command line input from the user until the number is multiple of 11.

Sample Input 1 :
Enter the number : 5
Expected Output format :5 is not multiple of 11. Try again.

Note : The command line input should not quit until the user enters a number which is multiple of 11. It should keep asking for a new input number.

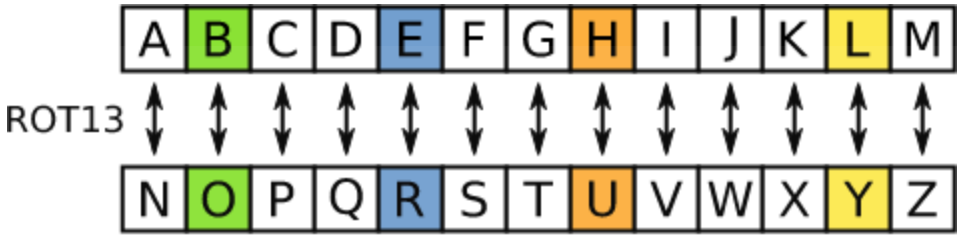
Solution

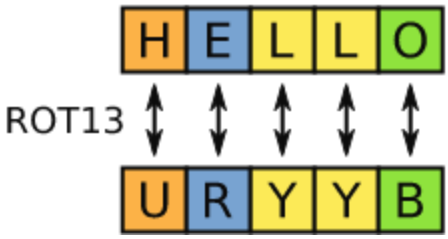
```
var readlineSync = require('readline-sync');
var num = null;

while (true) {
  num = Number(readlineSync.question('Enter Your Number :'));
  if (num % 11 === 0) break;
  console.log(`${num} is not multiple of 11.Try again.`);
}
```

Problem 3

Write an Algorithm to implement ROT13 ("rotate by 13 places", sometimes hyphenated ROT-13)

Description
About ROT13: Its a piece of text merely requires examining its alphabetic characters and replacing each one by the letter 13 places further along in the alphabet, wrapping back to the beginning if necessary.[2] A becomes N, B becomes O, and so on up to M, which becomes Z, then the sequence continues at the beginning of the alphabet: N becomes A, O becomes B, and so on to Z, which becomes M. Only those letters which occur in the English alphabet are affected; numbers, symbols, whitespace, and all other characters are left unchanged. Because there are 26 letters in the English alphabet and 26 = 2 × 13, the ROT13 function is its own inverse:[2]





Solution

```
const rot13 = (str) => {
      return str.replace(/[a-zA-Z]/gi, (s) => {
            return String.fromCharCode(
                  s.charCodeAt(0) + (s.toLowerCase() < 'n' ? 13 : -13),
            );
      });
};

const newStr = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';

console.log(rot13(newStr)); //
nopqrstuvwxyzabcdefghijklmNOPQRSTUVWXYZABCDEFGHIJKLM

console.log(rot13(rot13(newStr))); //
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

console.log(rot13('hello')); // uryyb

console.log(rot13('HELLO')); // URYYB
```

Problem 4
Write an Algorithm to accept a number as a command line input. Check for the number is a Prime Number or not. If it's a prime number, print the multiplication table of the number.
Sample Input 1 : 2
Sample Output 1 :
2 is a Prime Number.
2 x 1 = 2

2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
Sample Input 2 : 4
Sample Output 2 : 4 is not a prime number.

Note: The input number should be given as a command line. Program should keep asking for numbers until the user terminates with 'N'

Solution

```
var readlineSync = require('readline-sync');

//O(sqrt(n))
function isPrime(num) {
  for (let i = 2; i * i <= num; i++) {
    if (num % i == 0) {
      return false;
    }
  }
  return true;
}

//O(sqrt(n))
function printTable(num) {
  let prime = isPrime(num);
  if (prime) {
    console.log(`${num} is a Prime Number`);
    for (let i = 1; i <= 10; i++) {
      console.log(`${num} x ${i} = ${num * i}`);
    }
  } else {
    console.log(`${num} is not a Prime Number`);
  }

  return prime;
}

function getInputs() {
  let takeInputs = true;
  while (takeInputs) {
    let num = Number(readlineSync.question('Enter Your Number :'));
    if (!Number(num)) break;
    printTable(num);
    let input = readlineSync.question(
      "Press 'N' to Terminate or 'Y' to continue :"
    );
```

```
    if (input === 'N' || input === 'n') {
      takeInputs = false;
    }
  }
}

getInputs();
```

Problem 5
Write an Algorithm to take an input String ‘S’ with length ‘N’, split the string into two strings based on even and odd indexes while left padding the sub string with ‘000’ and right padding the substring with ‘111’.

Sample Input 1 :
code.in
Sample Output 1 :
000cd.n111
000oei111
Sample Input 2 :
Hello there
Sample Output 2 :
000Hlotee111
000el hr111

Note: The input number should be given as a command line. Program should keep asking for numbers until the user terminates with ‘N’

Solution

```
var readlineSync = require('readline-sync');

// O(n)
function splitAndPad(str) {
  let length = str.length;
  let even = '000';
  let odd = '000';

  for (let i = 0; i < length; i++) {
    if (i % 2 === 0) {
      even += str[i];
    } else {
      odd += str[i];
    }
  }

  console.log(even + '111');
  console.log(odd + '111');
}
```

```
function getInputs() {
  let takeInputs = true;
  while (takeInputs) {
    let str = readlineSync.question('Enter Your String : ');
    splitAndPad(str);
    let input = readlineSync.question(
      "Press 'N' to Terminate or any other character to continue : "
    );

    if (input === 'N' || input === 'n') {
      takeInputs = false;
    }
  }
}

getInputs();
```
